BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

IEEE TRANSACTIONS ON

COMPUTERS

JULY 1971 VOLUME C-20 NUMBER 7

Published Monthly

SPECIAL ISSUE ON MICROPROGRAMMING

[6] ILLIAC-IV System Study, Burroughs Corp., Univ. Illinois, Final Rep. 09852-B, 1966.
[7] "System 360 model 40, 2040 processing unit," in *IBM Field Engineering Diagrams Manual*, Doc. 0223-2842, 1966.
[8] S. G. Tucker, "Emulation of large systems," *Commun. Ass. Comput. Mach.*, vol. 8, Dec. 1965, pp. 753–761.
[9] R. W. Cook and M. J. Flynn, "System design of a dynamic microprocessor," *IEEE Trans. Comput.*, vol. C-19, Mar. 1970, pp. 213–222.
[10] V. R. Lesser, "Direct emulation of control structures by a parallel microcomputer," *Stanford Linear Accelerator Center*, Stanford Univ., Stanford, Calif., Rep. 127, Oct. 1970.
[11] ——, "A multi-level computer organization designed to separate data-accessing from the computation," Dep. Comput. Sci., Stanford Univ., Stanford, Calif., Tech. Rep. CS 90, 1968.
[12] S. Lass, "A fourth generation computer organization," in *1968 Spring Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 32. Washington, D. C.: Thompson, 1968, pp. 435–442.
[13] J. B. Dennis and E. C. Van Horn, "Programming semantics for multi-programmed computation," *Commun. Ass. Comput. Mach.*, vol. 9, Mar. 1966, pp. 143–155.
[14] J. J. Horning and B. Randell, "Structuring complex processes," *IBM Watson Res. Cen.*, Yorktown Heights, N. Y., Rep. RC-2459, 1969.
[15] M. E. Conway, "A multi-processor system design," in *1963 Fall Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 24. Baltimore: Spartan, 1968, pp. 139–146.
[16] *PDP-II Reference Manual*, Digital Equipment Corp., 1969.
[17] H. W. Bingham and E. W. Reigel, "Parallelism exposure and exploitation in digital computing systems," Burroughs Corp., Paoli, Pa., Final Tech. Rep., 1969.
[18] O. Dahl, B. Myhrhang, and K. Nygaard, "Simula 67, common base language," *Commun. Ass. Comput. Mach.*, vol. 9, 1966.
[19] B. W. Lampson, "Dynamic protection structures," in *1969 Fall Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 35. Montvale, N. J.: AFIPS Press, 1969, pp. 27–38.
[20] ——, "A scheduling philosophy of multi-processing systems," *Commun. Ass. Comput. Mach.*, vol. 11, May 1968, pp. 347–359.

Functional Memory and Its Microprogramming Implications

PETER L. GARDNER

Abstract—Functional memory (FM) is a general-purpose systems technology and has been proposed as a solution to the problems of large-scale integration. It is based on an associative array, composed of writable storage cells capable of holding three states; 0, 1, and DON'T CARE. The functional memory module can be used either as a local store, control store, associative store, or logic block. In its use as a logic block, logic is performed by associative table lookup, using the DON'T CARE state to give significant compression of tables over conventional two-state arrays (typically n to n^2 words for functional memory instead of 2^n words for conventional two-state arrays). The basic properties of functional memory are described in [1].

This paper is in two parts. The first part retraces some of the central material in [1], examines the reasons for table compression, and extends the techniques of table design. Following from a discussion of static tables, the treatment of time-dependent tables leads naturally to microprogramming. The second part identifies two problems with microprogramming and suggests ways to overcome them. The problems are branching delay and field combination. The branching delay problem is the delay inherent in using a data flow condition to modify the address of the next control word to be fetched. The solution is seen in the classification of data flow conditions into two classes: those that give only minor changes in direction (e.g., the kind of differences that exist between ADD, SUBTRACT, and COMPARE), and those that give gross changes in direction (e.g., differences between, say, ADD and EDIT); and in keeping the minor conditions in the data flow and allowing only the gross conditions to return to the control store. Conceptually, whereas the gross changes modify the *address* of the control word to be fetched, the minor changes modify the *contents* of the control word that has already been fetched.

The field combination problem occurs when the processing of two or more fully or semi-independent (micro) instruction streams is controlled by a single series of control words, each of which controls the whole data flow for a cycle. The problem is that the number of control words necessary is related to the *product* of the independent instances, and may therefore be very large.

The solution is seen in the provision of multiple control streams with linking only where necessary. The number of control words necessary would then be related to the *sum* of the independent instances, and should therefore be smaller.

In discussing these problems, functional memory is not seen as the solution, per se, but as clarifying the problems and making the solution easier to find. The flexibility of functional memory is also seen as a stimulus to the further development of microprogramming.

Index Terms—Array logic, associative memory, DON'T CARE state, functional memory, LSI, microprogramming, modular memory, table lookup.

Manuscript received October 12, 1970. This paper is published with the permission of the IBM Corporation.

The author is with the IBM, United Kingdom Laboratories, Ltd., Winchester, Hants, England.

PART I: FUNCTIONAL MEMORY AND LOGIC

A. Why Array Logic?

THE problems of large-scale integration (LSI) have been stated many times, e.g., [1], [2], and need not be repeated here. Suffice it to say that, because of these problems, the following properties are desirable for a logic module made in LSI:

1) high circuit density
2) regularity of interconnections between circuits
3) high circuit-to-edge connector ratio
4) loadable, i.e., have easily changeable personality; this is to ensure high usage of module type
5) easily testable.

The circuit organization that fits these requirements best is a storage array. Functional memory attempts to solve the problems of LSI by using a storage array to perform logic economically.

B. Conventional Array Logic

Traditional methods of performing logic on two operands by table lookup have been acceptable for narrow operand inputs, but have been prohibitively expensive for wide operand inputs because the number of words required has been related to 2^{2n}, where n is the operand width in bits (for example, related to 65 536 words for byte-wide operands). These methods have used either two-state directly addressed table lookup or two-state associative (content addressable) table lookup.

Assume that the Boolean OR function of bit pairs is to be performed on two two-bit operands, as represented in Fig. 1. The two-state directly addressed memory would require 16 words of two bits, as shown in Fig. 2. The two-state associative memory would require 15 words of six bits, as in Fig. 3, two bits for READ output, as before, plus four bits for SEARCH input in place of address decoder.

Note that, because the information stored in the table is not dependent on position within the memory, the search input 0000 can be omitted, since its READ output 00 is no different from the absence of output (unless checking is performed by a test on one and only one word selected).

C. Functional Memory Array Logic

In functional memory, where each cell position is capable of holding a binary 1 (matches a 1 input), a binary 0 (matches a 0 input) and a DON'T CARE (matches a 0 input or a 1 input), the same table compresses to four words, as in Fig. 4. The DON'T CARE state is represented by X.

Note that in functional memory, multiple selection of words is possible. The associated READ outputs are ORed to form the result.

For the OR of bit pairs of two operands the number of words required is the following.

Operand Width (bits)	Two-State Directly Addressed (words)	Two-State Associative (words)	Functional Memory (words)
2	16	15	4
4	256	255	8
8	65 536	65 535	16
n	2^{2n}	$2^{2n}-1$	$2n$

The number of words required varies per function. For example, the AND requirements are as follows.

Operand Width (bits)	Two-State Directly Addressed (words)	Two-State Associative (words)	Functional Memory (words)
2	16	7	2
4	256	175	4
8	65 536	58 975	8
n	2^{2n}	$2^{2n}-3^n$	n

It can be seen from these figures that the degree of table compression possible with functional memory can be very significant. Before looking at the effect of functional memory on other, more complex functions, let us establish what it is that gives functional memory this advantage.

D. Why Functional Memory Tables Compress

The characteristics of functional memory that allow the compression of logic tables are as follows.

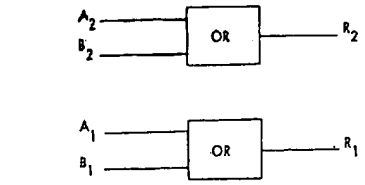

Fig. 1. Bit pair ORing of two two-bit operands.

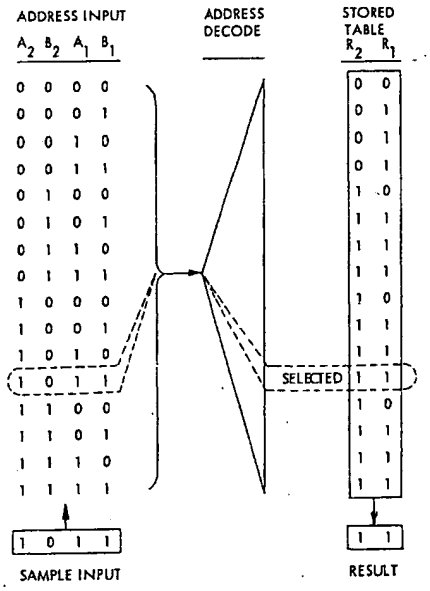

Fig. 2. Two-state directly addressed table lookup.

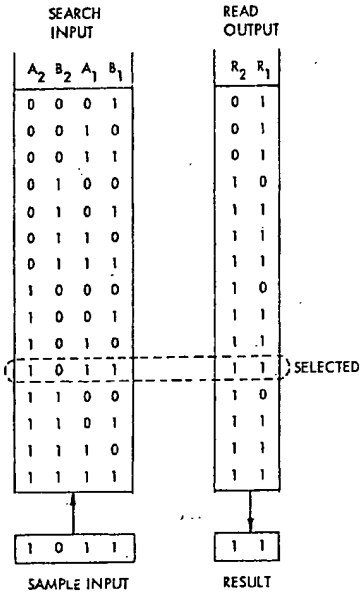

Fig. 3. Two-state associative table lookup.

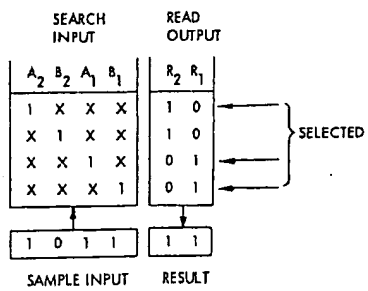

Fig. 4. Functional memory table lookup.

1) Independent subfields: Each result output bit is treated separately. Only those input bits which *should* contribute to a particular result output bit are *allowed* to contribute. All others are deliberately excluded. The result output bit with its relevant input bits are together regarded as a subfield and an independent table is built for each subfield. All subfields are processed simultaneously and the result output bits put together by concatenation. For instance, in Fig. 1, A_2, B_2, and R_2 can be regarded as a subfield and A_1, B_1, and R_1 as another. The table could be represented as in Fig. 5.

2) Combinations of interest: Within each subfield only those combinations of input bits that actually produce an output are included. Combinations producing a 0 output are excluded.

3) Compression of combinations to find prime implicants: The combinations of input bits that produce an output within a subfield are examined to find the prime implicants. For instance, if from inputs A, B, C, and D combinations $A \cdot B \cdot C \cdot \bar{D}$. and $A \cdot B \cdot C \cdot D$ both produce the same output, then only one entry need be made for the two combinations. $A \cdot B \cdot C$ is entered and the D position is stored as a DON'T CARE. This is the obvious use for the DON'T CARE state of the cell.

Not all of these three characteristics are exclusive to functional memory (or to any other three-state associative organization). The first, independent subfields, is possible with both two-state directly addressed memories and two-state associative memories, and it is this characteristic, out of the three, that probably gives the greatest table compression.

Subfields *could* be processed independently by two-state directly addressed memory and two-state associative memory by providing a separate, smaller array for each subfield and concatenating the result output bits. The unwanted subfields would therefore be DON'T CARE by omission. Thus the total number of words required for a function would then be related not to the *product* of the number of words required for all independent subfields (as at present), but to the *sum* (as for functional memory).

However, there would still remain the problem of deciding on the number of small arrays to provide (i.e., how many independent subfields to allow for) and on their input width and number of words. This decision would vary according to function and would have to be made at manufacturing time. With functional memory, however, because one DON'T CARE is provided for each cell position instead of one for each subfield, no such decisions have to be made, and it is this factor which makes functional memory attractive as a general-purpose logic block.

With associative arrays, whether two- or three-state, some of this effect could be achieved by providing a DON'T CARE per column (i.e., one per each bit of width). This, of course, is masking. The problem would remain, however, of deciding at manufacturing time on the number of words over which each mask should have its effect, i.e., the number of words required for each subfield. It is interesting to note that the DON'T CARE state of the functional memory cell is in effect a SEARCH mask per bit position.

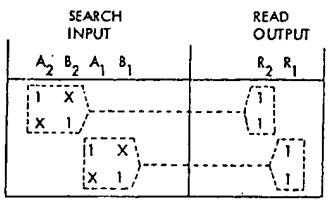

Fig. 5. Illustration of independent subfields.

The second characteristic, the inclusion of only those combinations of interest, is common to any associative memory, whether two- or three-state.

The third characteristic, the compression of combinations to find prime implicants, is a function of the DON'T CARE state of the *cell* (as opposed to the DON'T CARE state of the *subfield*) and is not shared by the two-state memories.

E. The Functional Memory Module

Fig. 4 showed the basic operation in functional memory whereby operands to be processed are presented as SEARCH input to the SEARCH input field of an array holding bit patterns appropriate to the function to be performed. The bits read from the READ output field are the result.

In functional memory a register, called an I/O (input/output) register, holds the SEARCH input during SEARCHing and the result during READing.

Although the SEARCH input field and the READ output field are shown as separated, there is no physical boundary between them inside the module. The SEARCH and READ fields can be of any width (within the limits of the module) and may overlap in any way. This gives generality and greater flexibility in choice of field widths than a fixed partition would allow. However, this generality must be paid for in two ways (see Fig. 6).

1) Two-phase cycle: The result of the SEARCH is set into a register of latches, called *selectors*. There is one selector per word in the array. The process of searching the SEARCH input field of the array from the I/O register and of setting the result into the selectors is called the SELECT phase.

The second phase is the READ/WRITE phase. During this phase, if READing, the selector contents are used to access the array a second time and the result is set into the I/O register.

The module cycle is made up of a SELECT phase followed followed by a READ/WRITE phase.

2) Masking: The SEARCH input field and the READ output field are defined for each cycle by a pair of masks, the SEARCH mask and the READ/WRITE mask, respectively. The mask pair required for a given cycle is chosen from a mask stack by a set of mask address bits presented to the module at the start of the cycle. The mask stack is resident in the module and is loaded, together with the array, at initial load.

The selector register is defined as a shift register, the shifting being performed at the end of the SELECT phase by the external control NEXT, which is provided to the module at the beginning of the cycle. There are two important uses for NEXT.

a) Loading: During initial load, words to be loaded cannot be addressed by content. NEXTing is an alternative method of addressing.

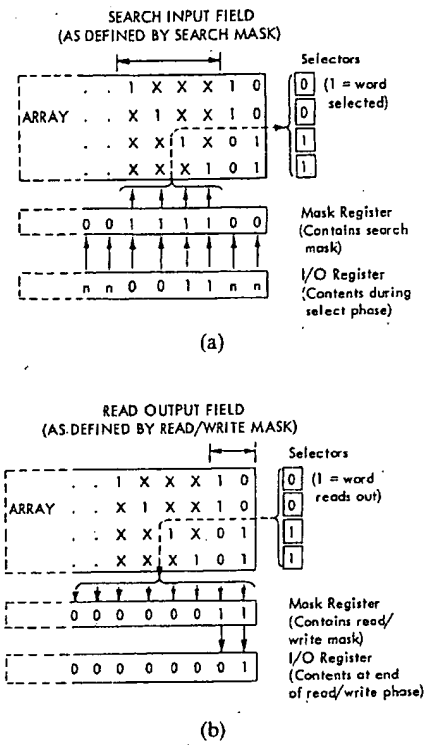

Fig. 6. (a) SELECT phase (SEARCH operation). (b) READ/WRITE phase (READ operation).

b) Word width: Should the total width of SEARCH input plus independent READ output exdeed the total word width available in the module, then the SEARCH input patterns and the READ output patterns can be interleaved such that the READ output field for each table "word" is placed "next" to its corresponding SEARCH input field.

After a READ operation the result of the current cycle either remains in the I/O register of the same module or is passed via the *data bus* to another module for accessing another table in the following cycle. The functional memory system is therefore in three parts, as in Fig. 7.

The I/O register communicates with the data bus by means of a set of bidirectional data pins. There is one data pin per bit position in the I/O register. Although only one I/O register and one set of data pins is necessary for the correct working of the functional memory module, it has been found that a significant improvement in array usage can be achieved by adding a second I/O register and a second set of data pins. The choice is made at the start of the cycle, under external control, as to which I/O register is to provide the SEARCH input and which is to receive the READ output. Pipelining through the array is thus possible.

In addition to the two sets of data pins, the module also has mask address pins (for addressing the mask stack), control pins (to control choice of I/O register for SELECT phase, choice of I/O register for READ/WRITE phase, whether to SEARCH, NEXT, READ, or WRITE, etc.), and pins to carry power and clock signals.

The data pins, mask address pins, and control pins all have compatible voltage levels on the bus. An FM data flow is composed of one or more FM modules with their data pins, mask address pins, and control pins connected together as required. Control of the data flow can either be from a control store or from the data flow itself. The provision of the control from the data flow itself is known as autosequencing and is a powerful technique in functional memory.

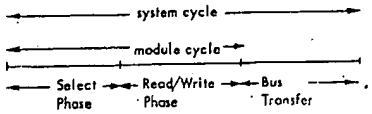

Fig. 7. Functional memory system cycle.

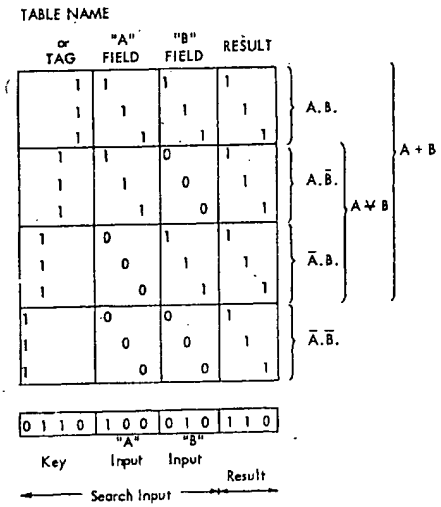

Fig. 8. Complete logical table.

Only those properties of functional memory necessary to understanding have been described above. Flinders *et al.* give a more detailed description of the module functions, together with a discussion of the checking, diagnosis, and recovery aspects.

F. Table Design in Functional Memory

Fig. 4 showed a functional memory table for performing a bit-pair OR of two two-bit operands. When two or more tables exist simultaneously in the same module, or in the same group of modules common-wired (known as a "store"), they are distinguished from each other by means of a table name or "tag." This is a field of bits (0, 1, or DON'T CARE), stored in the array together with the tables at initial load, which form part of the search input field and which is searched by a "key" during the SELECT phase. Thus the relationship between the key in the current cycle and the tags stored beside the tables defines the function to be performed on the operand(s) in the current cycle.

The tables in Fig. 8 (repeated from [1]) provide the four bit-pair products of two three-bit operands, $A_3A_2A_1$ and $B_3B_2B_1$, with their tags.

The tables can be used individually (for example, key 0010 will select table $A_n \cdot \bar{B}_n$ and exclude all others) or in combination to select any of the 16 possible logical functions of two variables. The contents of the I/O register in Fig. 8 show the key and operand input bits (together forming the SEARCH input) and the result output for an XOR operation.

For clarity, the figures illustrating tables hereafter use

INPUT				OUTPUT				
A_4	A_3	A_2	A_1	R_5	R_4	R_3	R_2	R_1
			0					1
		0	1				1	
		1	0				1	
	0	1	1			1		
	1	0				1		
	1		0			1		
0	1	1	1		1			
1	0				1			
1		0			1			
1			0		1			
1	1	1	1	1				

Fig. 9. Increment.

blanks to represent those cells holding the X state. Only cells holding a 1 read out as 1; cells holding a 0 or an X read out as 0.

In designing a table for a particular function, it is necessary to express each result bit (in true form) as the logical sum of products of the input bits, all factors having been removed, and then map the equations into table form.

For instance, if the four-bit operand $A_4A_3A_2A_1$ is to be incremented by one in its least significant position (A_1) and the five-bit result $R_5R_4R_3R_2R_1$ is to be output, we would need to express the function as

$$R_1 = \bar{A}_1 \tag{1}$$

$$R_2 = \bar{A}_2 \cdot A_1 + A_2 \cdot \bar{A}_1 \tag{2}$$

$$R_3 = \bar{A}_3 \cdot A_2 \cdot A_1 + A_3 \cdot \bar{A}_2 + A_3 \cdot \bar{A}_1 \tag{3}$$

$$R_4 = \bar{A}_4 \cdot A_3 \cdot A_2 \cdot A_1 + A_4 \cdot \bar{A}_3 + A_4 \cdot \bar{A}_2 + A_4 \cdot \bar{A}_1 \tag{4}$$

$$R_5 = A_4 \cdot A_3 \cdot A_2 \cdot A_1. \tag{5}$$

This is then mapped into a table such as Fig. 9. Note that incrementing by means of this table requires only one cycle, since carry ripple is dealt with in the table.

Note also that an increasing number of words need to be added to the table as each new bit of input width is added:

$$1 + 2 + 3 + 4 + \cdots n(+1 \text{ for carry}),$$

i.e., a total of

11	for a	four-bit operand
37	for an	eight-bit operand
137	for a	16-bit operand
$(n+1)\times n/2+1$	for an	n-bit operand.

Provided that it can be written as the logical sum of products, any function *can* be mapped into a one-cycle functional memory table. However, some functions require a large number of words (such as 137 words for the 16-bit increment); for some other functions (such as binary ADD), the number of words required is so large that one is forced to design tables for two or more cycles, the intermediate results from the first table being passed as input to the second table.

It is reasonable, therefore, to search for additional module functions that can increase the logical power of the module while remaining generally useful. READ RIGHT XOR LEFT (XOR means EXCLUSIVE OR) has been found to be such a function.

G. Functional Memory Cell and READ RIGHT XOR LEFT

Before illustrating the uses of READ RIGHT XOR LEFT in the design of functional memory tables, a brief discussion about the FM cell is necessary.

The functional memory cell is composed of two bistables, named "left bistable" and "right bistable," each connected to its own bit line ("left bit line" and "right bit line," respectively), and both connected to a common word line. The cell states are represented as follows.

Cell State	Left Bistable	Right Bistable
0	1	0
1	0	1
X	0	0
Y	1	1

(The Y state is not essential to the functional memory concept, but can be useful, as in the use of READ RIGHT XOR LEFT.)

A column consists of a left bit line and its corresponding right bit line together with all the bistables connected to them. A word consists of a word line together with all the bistables connected to it. Columns and word lines, when put together as an orthogonal matrix, form the functional memory array (see Fig. 10). Each column in the array is connected to a corresponding bit position in the I/O register.

During SEARCH the bit lines are driven according to the polarity of the corresponding I/O register bit and SEARCH mask bit for that cycle; thus we have the following.

I/O Register Bit	SEARCH Mask Bit	Left Bit Line	Right Bit Line
0	1	lower	raise
1	1	raise	lower
any	0	lower	lower

During SEARCH each bistable gates out a mismatch signal onto its word line if and only if it contains a one and its bit line is raised. Mismatch signals gated onto a given word line are ORed and the inverse of the OR is set into the corresponding selector to indicate "word selected."

During READ the selected word lines are driven and the bistables that are attached to them output onto their corresponding bit lines if and only if they contain a one and their word line is raised. The signals gated out onto a given bit line are ORed.

With the tables shown so far and in [1], the information ORed onto the left bit line was ignored during READ. The ORed information on the right bit line was set into the corresponding I/O register bit position (having been gated by the READ/WRITE mask). Hence only cell state = 1 had any effect on the result of a READ. (Cell state = Y read as = 1, and was therefore not used.)

With READ RIGHT XOR LEFT the ORed information on the

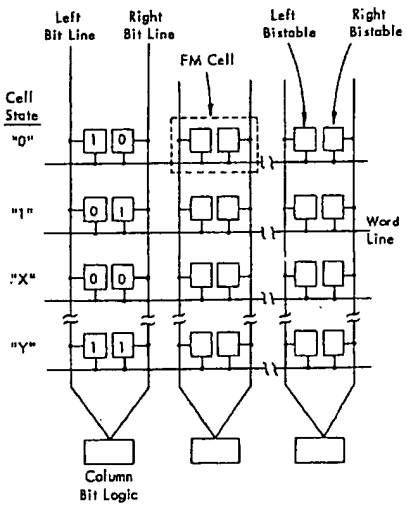

Fig. 10. Functional memory cell and array.

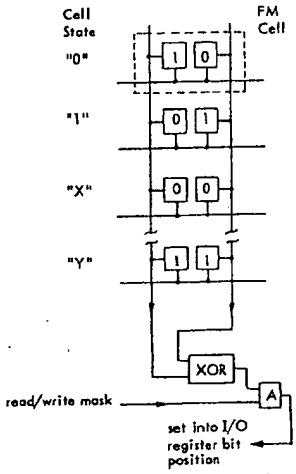

Fig. 11. READ RIGHT XOR LEFT on single functional memory column.

right bit line is XORed with the ORed information on the corresponding left bit line before being gated by the mask bit and set into the corresponding I/O register bit position. Thus cell state = 0 can now have an effect on the read result. Also, cell state = Y, if read out onto a column, can now be used to inhibit the output of that column (see Fig. 11).

H. The Module Function

With READ RIGHT ONLY the result bits were expressed as the logical sum of products

$$R_k = (A_1 \cdot B_1 \cdot C_1 \cdot \cdots \cdot N_1) + (A_2 \cdot B_2 \cdot C_2 \cdot \cdots \cdot N_2) + \cdots + (A_n \cdot B_n \cdot C_n \cdot \cdots \cdot N_n) \quad (6)$$

where each element K_i could be true (=1), complement (=0), or ignore (=X). This equation was true whether the table was in one module or spread over several with the outputs of each being dot-ORed on the data bus. In abbreviated notation the total module "function" could be expressed as

$$\cdot + \cdot + \cdot + \cdots + \cdot + \cdot \quad (7)$$

where · represents a product, i.e., one word's worth of matching and selection, and + represents the ORing of the READOUT either onto the bit lines (right bit line only) or onto the bus.

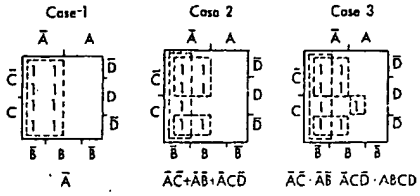

Fig. 12. Karnaugh maps for READ RIGHT ONLY.

Input				Output
A	B	C	D	Result
0		0		1
0	0			1
0		1	0	1
1	1	1	1	1

Fig. 13. Case 3 with READ RIGHT ONLY.

With READ RIGHT XOR LEFT the total module function can be expressed as

$$[\{(\cdot + \cdot + \cdots) \forall (\cdot + \cdot + \cdots)\} \cdot \rceil (\cdot + \cdot + \cdots)] + [\text{same}] + [\text{same}] + \cdots \quad (8)$$

(1, 2, 5, 3, 4, 6, 7)

where 1 represents a product, i.e., SEARCH and MATCH; 2 represents ORing of READOUT onto a given bit line; 3 represents ORed information on, say, right bit line; 4 represents ORed information on, say, left bit line; 5 represents RIGHT XOR LEFT, $\forall$ means EXCLUSIVE OR; 6 represents READing out the Y state contained in one or more cells in a given column; $\cdot \rceil$ means AND NOT; and 7 represents ORing of I/O register bits on the data bus.

The use of the Y state has been shown as separate in 6 (above) in order to show its effect more clearly. The physical operation is such that the logical sum of products in bracket 6 is actually included in brackets 3 and 4 at the same time.

I. READ RIGHT XOR LEFT Explained by Karnaugh Maps

The effects of READ RIGHT XOR LEFT can be demonstrated on Karnaugh maps in Fig. 12. Case 1 is "complete." Case 2 is "nearly complete." Case 3 is "nearly complete plus a bit more." In all cases READ RIGHT ONLY requires building up the function additively. For instance, Case 3 would map into a READ RIGHT ONLY table as the four words in Fig. 13.

Whereas with READ RIGHT ONLY the designer is obliged to build up the table additively, selecting only those prime implicants that he wants, with READ RIGHT XOR LEFT he has the option to build additively or to select more than is wanted and to remove the unwanted portions. For example, Case 1 remains unchanged since it is complete already. Cases 2 and 3, however, can be expressed as in Fig. 14. Both Case 2 and Case 3 would map into only two words each with READ RIGHT XOR LEFT, as in Fig. 15, contrasting with

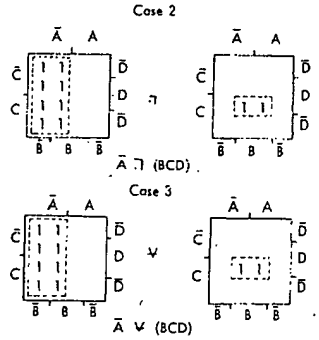

Fig. 14. Using READ RIGHT XOR LEFT. ⊼ means AND NOT; ⊻ means EXCLUSIVE OR.

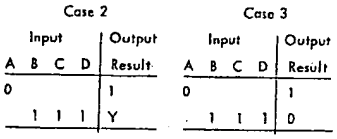

Fig. 15. Cases 2 and 3 with READ RIGHT XOR LEFT.

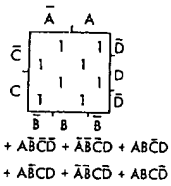

Fig. 16. Odd parity. READ RIGHT ONLY.

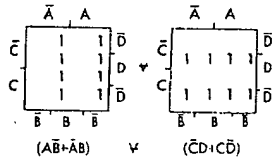

Fig. 17. Odd parity. READ RIGHT XOR LEFT.

the three and four words, respectively, with READ RIGHT ONLY. Note that in Fig. 15, Case 3, the result values 1 and 0 are interchangeable, provided that $\bar{A}$ outputs one value and $B \cdot C \cdot D$ outputs the other.

The most expensive function for the logical sum of products method of table mapping and the function for which READ RIGHT XOR LEFT is most obviously helpful is parity. This function can be represented as all the black squares on a chess board, as in Fig. 16. With READ RIGHT ONLY the table designer must select individually every input combination which produces a certain parity (say, odd). No compression is possible and within the subfield the X state is useless. With READ RIGHT XOR LEFT, however, odd parity could be represented, as in Fig. 17. The words required for a parity table are therefore the following where $n/2$ must be rounded up to the nearest integer.

(bits)	READ RIGHT ONLY (words)	READ RIGHT XOR LEFT (words)
4	8	4
8	128	16
n	2^{n-1}	$2^{n/2}$

J. Tables Using READ RIGHT XOR LEFT

The foregoing was an attempt to show the effects of READ RIGHT XOR LEFT in principle. Two examples should serve to show its effect in practice.

1) Increment: Four-bit operand $A_4A_3A_2A_1$ incremented by one: Equations (1)–(5) in Section F expressed each result bit as the logical sum of products, as required by READ RIGHT ONLY. With READ RIGHT XOR LEFT the equations can become

$$R_1 = \bar{A}_1 \tag{9}$$

$$R_2 = A_2 \veebar A_1 \tag{10}$$

$$R_3 = A_3 \veebar (A_2 \cdot A_1) \tag{11}$$

$$R_4 = A_4 \veebar (A_3 \cdot A_2 \cdot A_1) \tag{12}$$

$$R_5 = A_4 \cdot A_3 \cdot A_2 \cdot A_1. \tag{13}$$

These equations can map into a $2n$ word table, compared with $(n+1) \times n/2 + 1$ words required for READ RIGHT ONLY. For example, a 16-bit increment is now reduced from 137 to 32 words.

Although this reduction seems impressive, one can go further. Using the fact that $(a \veebar b) = (\bar{a} \veebar \bar{b})$, the equations for the increment result bits can now transform to

$$R_1 = \bar{A}_1 \tag{14}$$

$$R_2 = \bar{A}_2 \veebar \bar{A}_1 \tag{15}$$

$$R_3 = \bar{A}_3 \veebar (\bar{A}_2 + \bar{A}_1) \tag{16}$$

$$R_4 = \bar{A}_4 \veebar (\bar{A}_3 + \bar{A}_2 + \bar{A}_1) \tag{17}$$

$$R_5 = 1 \veebar (\bar{A}_4 + \bar{A}_3 + \bar{A}_2 + \bar{A}_1). \tag{18}$$

The table for increment now has the rather unexpected form given in Fig. 18.

Final reduction figures for increment are the following.

Operand Width (bits)	Two-State Directly Addressed (words)	Functional Memory READ RIGHT ONLY (words)	Functional Memory READ RIGHT XOR LEFT (words)
4	16	11	5
8	256	37	9
16	65 536	137	17
n	2^n	$(n+1) \times n/2 + 1$	$n+1$

2) One-cycle compare: Comparison of two operands, $A_4A_3A_2A_1$ and $B_4B_3B_2B_1$, for $A > B$ can be expressed as

$$A > B = A_4 \cdot \bar{B}_4 + A_3 \cdot \bar{B}_3 \cdot (A_4 + \bar{B}_4) + A_2 \cdot \bar{B}_2 \cdot (A_4 + \bar{B}_4) \cdot (A_3 + \bar{B}_3) + A_1 \cdot \bar{B}_1 \cdot (A_4 + \bar{B}_4) \cdot (A_3 + \bar{B}_3) \cdot (A_2 + \bar{B}_2). \tag{19}$$

With READ RIGHT ONLY, the expression must be multiplied out and $2^n - 1$ words are required for a one-cycle compare (where n is the width of each comparand). An additional $2^n - 1$ words are required for $A < B$, clearly unacceptable for medium to wide comparands.

With READ RIGHT XOR LEFT the expression can be transformed to

$$A > B = A_4 \cdot \bar{B}_4 + A_3 \cdot \bar{B}_3 \cdot \barwedge(\bar{A}_4 \cdot B_4) + A_2 \cdot \bar{B}_2 \cdot \barwedge(\bar{A}_4 \cdot B_4 + \bar{A}_3 \cdot B_3) + A_1 \cdot \bar{B}_1 \cdot \barwedge(\bar{A}_4 \cdot B_4 + \bar{A}_3 \cdot B_3 + \bar{A}_2 \cdot B_2) \tag{20}$$

where $\cdot \barwedge$ means AND NOT.

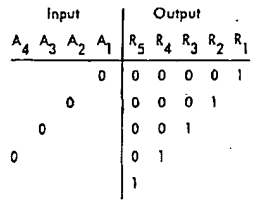

Fig. 18. Increment. READ RIGHT XOR LEFT.

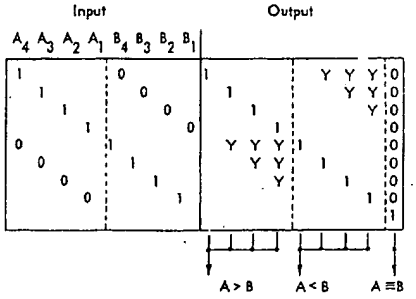

Fig. 19. Compare. READ RIGHT XOR LEFT.

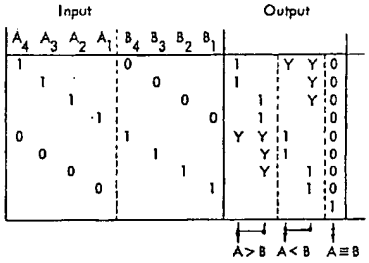

Fig. 20. Compare. Reduced width.

By allowing a separate output column for each major conjunct, and by dot-ORing those columns on the data bus, the $2n+1$ word table in Fig. 19 will provide not only the required $A > B$, but $A < B$ and $A \equiv B$ as well.

The tradeoff is thus a small (linear) number of columns against a large (2^n) number of words. Nevertheless, although the tradeoff is clearly worthwhile, word width is valuable. Therefore the following refinement should also be adopted.

Using the fact that $A_k \cdot \bar{B}_k$ can also be expressed as $A_k \cdot \bar{B}_k \cdot \neg(\bar{A}_k \cdot B_k)$, (20) can be reexpressed as

$$A > B = (A_4 \cdot \bar{B}_4 + A_3 \cdot \bar{B}_3) \cdot \neg(\bar{A}_4 \cdot B_4) + (A_2 \cdot \bar{B}_2 + A_1 \cdot \bar{B}_1) \cdot \neg(\bar{A}_4 \cdot B_4 + \bar{A}_3 \cdot B_3 + \bar{A}_2 \cdot B_2) \quad (21)$$

and the table in Fig. 19 will reduce in width to that in Fig. 20.

PART II: FUNCTIONAL MEMORY AND CONTROL

A. Introduction

The discussion so far has been about the logical aspects of functional memory—how tables can be designed to perform required functions. Most of the functions have been performed in a single cycle. However, some functions, such as binary addition, may require two or more cycles for the final result to emerge.

This introduces the idea of time-dependent tables. The second table must wait until initiated by the output of the first table. Alternatively, the second table can be initiated after a certain interval of time (i.e., a certain number of cycles) relative to a reference point. In both cases the stimulus is fed either to the table as part of the SEARCH input or to one of the external control lines of the module. A functional memory data flow is a collection of modules holding single cycle or sequenced tables (together with locations used for data storage, status information, etc.), joined together on the data bus as required. The sequencing of the use of the tables is performed by microprogram.

The microprogram of a functional memory data flow could be in a conventional control store with conventional sequencing and with the functional memory external controls and table keys each having their own control fields in the control word. This would make a reasonable system, the functional memory having replaced the logic of a conventional data flow.

But what would be the effect if the microprogram itself were in functional memory? It *could* be used to provide a conventional control store organization, as in the previous paragraph. The sequencing could be explicit—some of the bits read out in the current cycle being used to address the immediate successor by content-addressing, or implicit—using NEXTing to address the successor by position. Two control words could be stored in each functional memory word and distinguished from each other by READ RIGHT XOR LEFT through an appropriate filter of all 1's or all 0's, or by using READ modes RIGHT ONLY and LEFT ONLY.

However, using functional memory in this manner would be an expensive way of providing a conventional control store (the extra expense might be justified by other considerations, such as wanting a unified procedure for loading, checking, diagnosis, recovery, debug, etc.). Furthermore, it would also be to ignore the interesting implications that functional memory has for microprogramming.

Only two of these implications are discussed here. These concern the branching delay problem and the field combination problem. In this discussion functional memory is seen not as the solution to the problems per se, but as clarifying the problems and making solutions easier to find.

B. Branching Delay Problem

The conventional procedure for a microprogram branch is for a condition or conditions from the data flow to be passed to a black box to modify in some way (it may be a transform or merely concatenation) the address produced at the end of the preceding cycle. The control for the current cycle cannot start until the modified address is formed and the required control word accessed. This means either that the data flow and control store cannot be run at full speed and must be interleaved to some extent [see Fig. 21(a)], or that leap-frog branching must be accepted [see Fig. 21(b)]. In both cases delay is involved when an immediate branch is required.

This section does not attempt to eliminate branches that cause this delay, but to show that their inclusion in the microprogram is often unnecessary and can usually be avoided. In computer processing, decision making and branching are not synonymous. Consider three cases.

Case 1: When, in a conventional data flow, register P

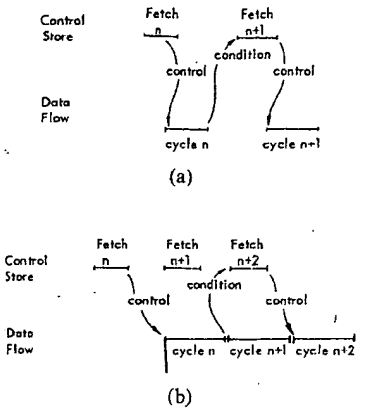

Fig. 21. (a) Control store and data flow interleaved. (b) Leap-frog branching.

and Q are controlled to provide their contents to an ALU (arithmetic and logic unit) to be added together, no branching need take place while the data pass through the ALU. Nevertheless, decisions are being made in the logic of the ALU; in other words, there is a difference between the way the logic behaves when, say, $P=3$ and $Q=5$ and when $P=3$ and $Q=6$. In this case it is most unlikely that the microprogram would branch.

Case 2: A further input to the ALU is the carry in. Its value affects the decision making within the ALU in the same way as the values of P and Q. If the ALU is narrower than the data widths then the data must be sectioned and the carry out of each section must be entered as carry in to the successor section. In this case it is quite possible that the decision to enter the carry or not might be made by a microprogram test and branch.

Case 3: Depending on the programmer instruction being performed, the control to the ALU may require to be add (as for ADD) or subtract (as for SUBTRACT or COMPARE) or some other function (e.g., as for LOGICAL INSTRUCTIONS). In this case it is very likely that the decision will be made by a microprogram test and branch.

As presented, these three decisions differ in their "level" within the machine control, and hence in the degree of probability that, in a conventional data flow, they will be treated by microprogram test and branch.

In Case 1 the microprogram might read "take the values found in registers P and Q and add them together." Parameters are being used, in this case P and Q.

A (micro)program might be described as a sequence of steps to be taken by the unit being controlled, using parameters the values of which are not known when the (micro) program is written. It is the use of parameters instead of exact values that allows for variety and decision making within the same basic "shape."

There is nothing fundamentally different between Cases 1, 2, and 3. All three could use parameters and avoid branching. An example might read "If condition C exists, do function specified by value in F (register or data bus) to contents of register P and Q."

Whereas branching involves taking conditions from the data flow and returning them to the control store to modify the *address* of the succeeding control word, the use of parameters within a single shape could be regarded as keeping the conditions in the data flow to modify the *contents* of the succeeding word.

Thus, if parameters are used, the following is true.

1) The succeeding control word can be fetched before the value of the condition is known. This saves time.

2) The same control word will do for two or more different paths within the same basic shape. This saves control bits.

The system designer should pitch the level of his microprogram language to suit the system architecture according to the following principle.

Principle 1: Data flow conditions are of two types: those that give only minor changes in direction (e.g., the differences that exist between ADD, SUBTRACT, and COMPARE) and those that give gross changes in direction (e.g., differences between, say, ADD and EDIT). Minor conditions should be kept in the data flow; only gross conditions should be allowed to return to the control store.

Although any data flow organization can apply this principle, functional memory is particularly well suited to doing so. Much of the *function* performed in a functional memory data flow is already more diffused than in a conventional microprogram data flow. Now much of the *control* can also be integrated into the data flow. Decisions are taken close to both the place generating the conditions on which the decisions depend and the place where the action must be taken.

In this respect functional memory forms an interesting bridge between conventional microprogram-controlled and hardware-controlled machines, combining many of the advantages of both.

C. Field Combination Problem

Although not fundamentally different from each other, program and microprogram tend to differ in one important respect. In general, program is concerned with only one facility or element of the system at a time; microprogram can be concerned with several at the same time.

Although in microprogram all the facilities or elements of the data flow may be contributing to the overall goal (for example, the particular programmer instruction being microprogrammed), the elements are often working in parallel on portions of the microprogram flow that can be considered independent of each other for the time being (say, for the following n cycles).

The truth of this is most obvious in a pipelined organization where there may be, say, three different programmer instructions in different stages of decomposition in the data flow at any one time. It is equally true, however, in a normal single-instruction stream, for instance with a System/360 RX instruction, where an operand from mainstore is processed with an operand from a programmer register. From the architecture point of view, there is no constraint on the order in which the operands are fetched: operand 1 could be fetched first, operand 2 could be fetched first, or both operands could be fetched at the same time. System/360

Model 40, for example, fetches both at the same time.

The control sequence for fetching operand 1 is therefore logically independent of the control sequencing for fetching operand 2, and this independence continues until the point where they are brought together for processing. However, unless each of the independent control sequences has its own independent addressing structure, the conventional single-addressing, single-control store organization can provide only pseudo-independence, and that at the expense of combination.

Consider a conceptual data flow for a CPU such as that shown in [1, Fig. 17] and repeated here as Fig. 22. It was composed of five boxes and the interconnections between them which need not concern us here. The main functions of the five boxes were as follows.

1) Address store (functional memory): forms and increments addresses for main store.

2) Work store (functional memory): acts as ALU and op decoder.

3) Local store (functional memory): holds programmer's registers and provides work space.

4) Main store (conventional).

5) Control store (assume conventional).

The data flow could be divided, for control purposes, into three functional areas which we will call

A main store and addressing

W ALU

L registers and data.

In a conventional control store, each word would control all three functional areas for one cycle by having a control field devoted to each functional area and the appropriate values in each of the control fields, as in Fig. 23. Each field will have a set of values used:

$$A_1;\quad A_2;\quad A_3;\cdots A_n$$
$$W_1;\quad W_2;\quad W_3;\cdots W_n$$
$$L_1;\quad L_2;\quad L_3;\cdots L_n.$$

It is likely that most of these values will appear many times in different combinations:

$$A_1, W_1, L_3$$
$$A_1, W_7, L_5$$
$$A_2, W_7, L_3, \cdots.$$

Since one control word is necessary for each combination, the total number of control words necessary in the control is related to the *product* of the number of different values used in each field. This is seen as the field combination problem.

If, on the other hand, the control fields were to be separated from each other and each field given a separate control store, then the total number of control words necessary would be related not to the *product* of the number of different values used in each field, but to their *sum*.

Let us extract a second principle.

Principle 2: The system designer should attempt to identify those areas of control which, though concurrent, are architecturally independent of each other and provide a separate source of control for each independent area for the duration of their independence.

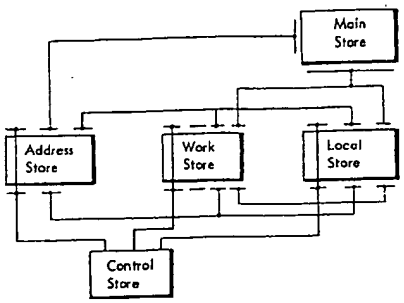

Fig. 22. A data flow.

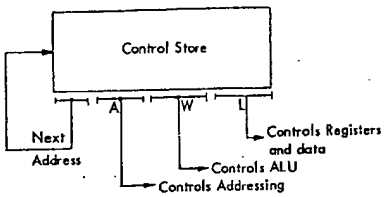

Fig. 23. Control fields.

There are several arrangements for providing independence.

1) Each control store full width (Fig. 24): Each control store is capable of controlling the whole data flow, but during any one cycle may control only one of the functional areas, thus we have the following.

	Control Store 1			Control Store 2			Control Store 3		
Cycle 1	O	W	O	A	O	O	O	O	L
Cycle 2	A	O	O	O	O	L	O	W	O
Cycle 3	A	O	O	O	W	O	O	O	L
etc.									

2) Each control store only field width (Fig. 25): Each control store is capable of controlling only one functional area of the data flow in any one cycle. The functional area to be controlled is determined by a field code which is read out with the field control value.

3) Control store per functional area (Fig. 26): Each control store is dedicated to a particular functional area. As data pass out of the control of one functional area, so too does addressing information from the control store of that functional area to the control store of the functional area to which the data are being passed. Each control store is thus sequenced not only by its own next address field, but also by next address information from the other control stores.

Of these three arrangements, the first and third seem of particular interest. The second arrangement would require an additional stage of decoding to decide which functional area is to be controlled. It is therefore not recommended.

An interesting difference, which can be illustrated in a pipeline organization, exists between the first and third arrangements. In the first, each control store could be devoted to a programmer instruction and could control it through the data flow from start to finish. (Clashes between instruc-

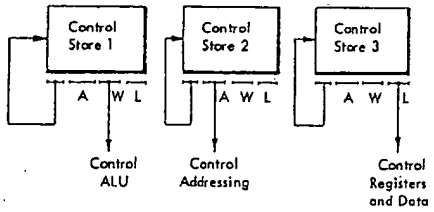

Fig. 24. Multiple control stores, full width.

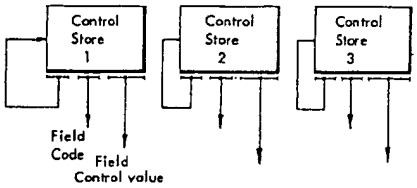

Fig. 25. Multiple control stores, field width.

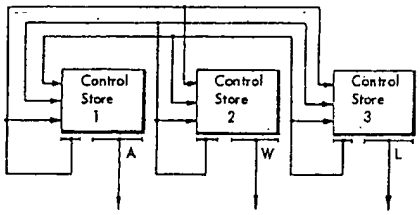

Fig. 26. Control store per functional area.

tions which cause holdoffs would presumably be detected in the data flow and returned to the control stores as holdoff or branching information.) There would, therefore, be as many control stores as the maximum number of instructions in the pipeline at any one time. With arrangement 3, on the other hand, since each control store controls a particular functional area, responsibility for each instruction is passed from one control store to another (one or many) as the information relative to that instruction passes around the data flow.

Although the conventional two-state directly addressed stores could be used to implement these control store arrangements, functional memory is particularly well suited to them. First, in its use in the data flow, functional areas are easier to define and isolate with functional memory than with conventional logic. This is because the elements of a conventional data flow tend to be defined according to their *logic* specialization (ALU, shifter, staticizers, etc.). With functional memory, on the other hand, logic function can be more diffuse and can be assigned to functional areas according to *architectural* requirements. Thus, for example, functional area *A* (main store and addressing) can contain tables for address formation, incrementing, and address specification testing, as required, and also locations for temporary storage of addresses. Second, in its use for control, arrangement 3 is well suited to the idea of integration of control into the data flow, as suggested in the discussion on the branching delay problem. However, it is in arrangement 1 that the implications of functional memory on microprogramming can be seen most clearly, even if not to best advantage (a mixture of arrangements 1 and 3 would probably be optimum).

It may seem wasteful to have only one third of each control store width active in any one cycle, but let us recall the discussion in Part I–D on the compression of tables. It was said there that probably the greatest degree of table compression results from the provision of independent logic subfields. What is being provided here by arrangement 1 is independent control fields, with no decision having to be made concerning the placing of those control fields.

It was suggested for arrangement 1 that each control store would control only one functional area in any one cycle. A natural extension is that in any one cycle any control store could control any number of functional areas from 0 to 3; thus we have the following.

	Control Store 1			Control Store 2			Control Store 3		
Cycle 1	*O*	*W*	*O*	*A*	*O*	*O*	*O*	*O*	*L*
Cycle 2	*A*	*O*	*L*	*O*	*O*	*O*	*O*	*W*	*O*
Cycle 3	*O*	*O*	*L*	*A*	*W*	*O*	*O*	*O*	*O*

A further extension is to place all three control stores into one functional memory store, and read out up to three words at the same time. There could then be a *variable* number of control stores, the number provided depending on the number of independent functional areas required from cycle to cycle. For instance, in the System/360 RX example, a single control sequence, with only a single word being read out each cycle, might precede the fetching of the operands. At that point the control could split into two independent sequences, two words being read out for each cycle, until they came together again as a single sequence for processing.

In these examples the control fields being read out in a cycle are put together by concatenation. Yet a further extension is possible if the control fields are put together in some other way, for example by ORing the fields or XORing using the READ RIGHT XOR LEFT.

Conclusion

Functional memory is basically an array. The circuits are therefore densely packaged and the interconnections between them are regular.

The array is associative and the cells are able to hold a DON'T CARE state in addition to the normal binary 0 and 1. Functional memory is therefore able to perform logic by table lookup very much more cheaply than conventional two-state table lookup methods. Techniques like READ RIGHT XOR LEFT provide even greater table compression.

Logic personality is loadable and easily changed. Module manufacture is therefore independent of the function that it is to perform in a system. Because of this independence, because it is an array, and because it can perform logic, functional memory is an attractive way of applying LSI to the field of random logic.

Functional memory words can be written during processing. It can therefore be used for local storage, working registers, status information, etc.

A functional memory data flow consists of a collection of

functional memory modules connected together. Because the elements of the data flow are all operating synchronously, are all of the same type, and are distinguished from each other only by their position in the data flow and by their contents, the system designer has the following option: either to specialize the elements by logic function (as with a conventional data flow) or to diffuse logic function around the data flow and form "functional areas" that are related to architectural requirements. He will probably do both.

The voltage levels of the data pins and external control pins of a functional memory module are compatible. Therefore "auto-sequencing" is possible, whereby some functional memory modules control others or themselves. Again the system designer has the option of controlling his data flow by a separate control store (as with a conventional microprogram-controlled data flow), or integrating the control into the data flow and providing independent local control where it would be advantageous. He will probably do both.

Thus functional memory not only seems a good solution to the problem of applying LSI to random logic, but also opens up new possibilities in the field of microprogram control.

ACKNOWLEDGMENT

The author wishes to acknowledge ideas contributed by numerous colleagues.

REFERENCES

[1] M. Flinders, P. L. Gardner, J. F. Minshull, and R. J. Llewelyn, "Functional memory as a general purpose systems technology," presented at the 1970 IEEE Computer Group Conference, June 1970.
[2] J. P. Bartlett, "Processing memories," presented at the 1970 IEEE Computer Group Conference, June 1970.
[3] B. T. McKeever, "The associative memory structure," in *Proc. Fall Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 27. Washington, D. C.: Spartan, 1965.

A Microprogrammed Intelligent Graphics Terminal

WILLIAM L. SCHILLER, ROBERT L. ABRAHAM, RICHARD M. FOX, AND ANDRIES VAN DAM, MEMBER, IEEE

Abstract—This paper describes a small computer, Interdata Model 3, that has been microprogrammed to serve as an intelligent terminal. The Interdata is connected to a System/360 multiplexor channel with a high-speed interface, and uses an ARDS direct view storage tube as a display console. The new Interdata target machine is patterned after the /360 (including all five instruction formats), but also has instructions particularly designed for intelligent terminal programming. These include instructions for character string manipulation, code conversion, list processing, coordinate manipulation, and virtual addressing. A powerful multiplexor channel, which allows the programmer to "overlap" I/O to several devices with a CPU program, has also been microprogrammed.

A program to simulate an IBM 2250 display console has been written for the Interdata. Although a 2250 is not an intelligent terminal, the simulator allows existing graphics programs at Brown University to be run using the Interdata-ARDS without any modifications.

The intelligent terminal will be used to investigate the appropriate division of labor between a central processor and satellite computer. Divisional tradeoffs will be studied by allowing the user to transfer routines of an applications program between processors at run time as a function of loading and response time.

Index Terms—Instruction set, intelligent graphics terminal, microprogramming, minicomputer, multiplexor channel.

Manuscript received October 30, 1970; revised March 2, 1971. This work was supported in part by NSF Grant GJ-181.

W. L. Schiller was with Brown University, Providence, R. I. He is currently with The Mitre Corporation, Bedford, Mass.

R. L. Abraham, R. M. Fox, and A. van Dam are with Brown University, Providence, R. I.

I. INTRODUCTION

A. Intelligent Terminals

OVER the last five years one of the main research topics in computer science at Brown University has been computer graphics. An active research and development group has completed a number of interactive applications programs, including the Hypertext Editing System, 3D-PDP (a program to lay out and analyze industrial piping systems), and the Flowchart Programming System. In all cases a buffered IBM 2250/Model 1 CRT, directly connected to a large System/360 computer, was used for interaction.

Recently we have become interested in replacing the 2250/Model 1 with a so-called "intelligent terminal." Such a terminal combines display facilities with the ability to do a significant amount of local processing. The utility of local processing is to relieve the main computer of vast amounts of I/O for low-level interaction and to provide the user with immediate response to modest computational or data manipulation requests. One intelligent terminal configuration which we have tested was a 2250/Model 4 connected to an IBM 1130, which in turn was interfaced over a 40.8 kilobaud line to the /360. This combination was on the one